Comments on Simple Statistical Functions (p1708r4): Contracts, Exceptions and Special cases

Johan Lundberg

Document #: P2376R0
Date: 2021-05-11
Project: Programming Language C++
Audience: SG19 (Machine Learning), SG6 (Numerics)
Email: lundberj@gmail.com

Abstract

p1708r4 proposes throwing a new exception `stats_error` to report error conditions. This paper reviews the the proposed features with focus on contracts, exceptions and special cases.

Guided by behaviour existing functions (such `std::log`, `std::sqrt`, `std::pow`), and libraries and tools for C++ and other languages, a specification scheme without exceptions is proposed.

The discussions relate to p1708r4 features mean, mode, skewness, kurtosis, variance, standard, deviation, including variations such as weighted mean. In addition, p1708r4 proposes quantiles and medians which are different enough and are not considered in this text.

I'm in favour of the effort of p1708r4 and would like to thank the authors and SG19 for pursuing it.

Contents

Contents 1
1 Introduction . . . 2
2 Mismatching range sizes (⋄) . . . 2
2.1 Conclusion . . . 2
3 Mathematical poles (†) and domain errors (∉) . . . 2
3.1 Conclusion . . . 3
4 Degenerate cases due to lack of data (∅) . . . 3
4.1 Matlab and Octave . . . 4
4.2 Boost math_toolkit . . . 4
4.3 C: Gnu Scientific library . . . 4
4.4 C#: MathNet.Numerics.Statistics . . . 4
4.5 R: MatrixStats . . . 5
4.6 Excel, Google Sheets . . . 5
4.7 Python: scipy.stats and numpy . . . 5
4.8 Pandas, Stata and SAS . . . 5
4.9 `std::log`, `std::sqrt` and `std::pow` . . . 6
4.10 Conclusion . . . 6
5 Other benefits of removing exceptions from the proposed Simple Statistical Functions 6
6 Notes on Non-floating point input . . . 6
6.1 Python and C++: generic algorithms and duck typing versus floating point arithmetic . . . 7
6.2 Non-Integer, non-floating point input . . . 7
7 Conclusion . . . 7
8 Appendix: Division by zero – NaN and exceptions in C#, Java, Python . . . 8

1 Introduction

p1708r4 proposes specifying throwing a single new exception `stats_error` to report error conditions. For example, for `mean`;

> (1) If r or w is empty$^{\varnothing}$ (or the sum of w is 0)† or r and w are of different sizes$^{\diamond}$, `stats_error` is thrown. In the case of `geometric_mean`, if any element of r is negative$^{\not<}$ or an element, along with its associated weight, is $0^{\dagger}$,stats error is thrown. In the case of `harmonic_mean`, if any element of r is $0^{\dagger}$,stats error is thrown.

2 Mismatching range sizes ($^{\diamond}$)

Considering (1)$^{\diamond}$, a similar situation to weighted means with r,w is `std::inner_product` or two-range `std::transform` which both demand that the second range is at least as long as the first one as part of the *contract*.

Violating the contracts of `std::inner_product` and `std::transform` is UB and can be dealt with by implementations by means of mechanism like checked iterators. Another possibility is to report contact violations like `std::condition_variable::wait` does and call `std::terminate`.

2.1 Conclusion

I find no reason for using exceptions to signal this kind of contract violations for the proposed simple statistical functions.

3 Mathematical poles (†) and domain errors ($^{\not<}$)

The mathematical definition and the numerical implementations of the proposes functions easily encounter poles and other numerical difficulties. Good examples are division by zero in kurtosis or skewness of all-equal elements; $\sigma = 0$. Examples are seen in (1)†.

What mathematical pole situations do we have in the current standard? Division by zero is undefined behaviour except for floating-point division where we get an implementation-defined value, and with IEEE we get specified results and behaviour[1].

— If one operand is NaN, the result is NaN dividing a non-zero number by ±0.0 gives the correctly-signed infinity and FE_DIVBYZERO is raised
— Dividing 0.0 by 0.0 gives NaN and FE_INVALID is raised

Similarly, `std::log`, (I'm here relying on cppreferences interpretation of the C++ standard including C99)

— If no errors occur, the natural (base-e) logarithm of arg (ln(arg) or log e(arg)) is returned.
— If a domain error occurs, an implementation-defined value is returned (NaN where supported).
— If a pole error occurs, -HUGE_VAL, -HUGE_VALF, or -HUGE_VALL is returned.
Error handling
— Errors are reported as specified in *math_errhandling*
— Domain error occurs if arg is less than zero.
— Pole error may occur if arg is zero.

[1] This is also the case in C# and Java and Pyton numpy. See appendix, section 8

If the implementation supports IEEE floating-point arithmetic (IEC 60559)

— If the argument is ±0, −inf is returned and FE_DIVBYZERO is raised.
— If the argument is 1, +0 is returned
— If the argument is negative, NaN is returned and FE_INVALID is raised.
— If the argument is +inf, +inf is returned
— If the argument is NaN, NaN is returned

`math_errhandling`[2] list the four[3] required floating point error conditions which can be queried using `std::fetestexcept`

3.1 Conclusion

The examples (1)† protects against pole errors using exceptions. I find no reason to use exceptions to signal poles and similar mathematical or numerical situations for mathematical functions. Instead it would be beneficial to specify the new functions in analogy with the existing mathematical functions: if a pole error occurs the results is unspecified unless IEC 60559 where NaN or ±Inf is returned along with the above mentioned math_errhandling.

Let's also consider the case of negative weights, (1)⁰. It is very similar to the domain error of `std::sqrt` of negative values. As such, it's better handled as a floating point domain error in analogy with the previous sections, than with exceptions.

4 Degenerate cases due to lack of data (⌀)

Both (1)⌀ and the case of (sample) variance are examples of where p1708r4 stipulate exception is thrown when the input is zero, one or two elements respectively.

Existing standard algorithms typically handle degenerate cases transparently and gracefully. The design of even `std::adjacent_difference` which could be thought to require at least two elements work naturally with one– and even zero sized input. Python `numpy.diff` does something similar and maps both single and empty data to an empty result.

Let's consider how these cases are handled in other libraries and languages.

[2]cppreference math_errhandling: https://en.cppreference.com/w/cpp/numeric/math/math_errhandling

[3]Error conditions

Domain error

the argument is outside the range in which the operation is mathematically defined (the description of each function lists the required domain errors) EDOM FE_INVALID std::acos(2)

Pole error

the mathematical result of the function is exactly infinite or undefined ERANGE FE_DIVBYZERO std::log(0.0), 1.0/0.0.

Range error due to overflow

the mathematical result is finite, but becomes infinite after rounding, or becomes the largest representable finite value after rounding down ERANGE FE_OVERFLOW std::pow(DBL_MAX,2).

Range error due to underflow

the result is non-zero, but becomes zero after rounding, or becomes subnormal with a loss of precision ERANGE or unchanged (implementation-defined) FE_UNDERFLOW or nothing (implementation-defined) DBL_TRUE_MIN/2.

Inexact result

the result has to be rounded to fit in the destination type.

4.1 Matlab and Octave

Matlab gives mean and variance[4] of [] to be NaN, and variance of a single element is 0.

Short input is gracefully allowed with conservative NaN results[5]

```
Matlab kurtosis [] is NaN
Matlab kurtosis [1] is NaN
Matlab kurtosis [1,2] is NaN
Matlab kurtosis [1,2,3] is NaN
Matlab kurtosis [1,2,3,4] is 1.64
```

Octave is a free alternative to Matlab and is similar but not identical. It gives mean and standard deviation of [] to be NaN, but variance of a single element is 0 instead of NaN.

Returning zero variance instead of NaN has a clear down side in that it can easily be misinterpreted as good data, eg `assert(variance<lim);`.

Further

```
octave kurtosis [] generates a runtime error
octave kurtosis [1] is NaN
octave kurtosis [1,2] is 2
octave kurtosis [1,2,3] is -1.5
octave kurtosis [1,2,3,4] is 1.64
```

4.2 Boost math_toolkit

Boost univariate statistics mean[6] does not specify clearly, but there mean of [] gives NaN and sample variance of [1] is 0.

4.3 C: Gnu Scientific library

Gnu Scientific library (GSL) reports errors[7] of kinds similar to c99, but has return codes, plus the possibility to register an error handler. The two most relevant error types are:

GSL_EDOM

Domain error; used by mathematical functions when an argument value does not fall into the domain over which the function is defined (like EDOM in the C library)

GSL_ERANGE

Range error; used by mathematical functions when the result value is not representable because of overflow or underflow (like ERANGE in the C library)

4.4 C#: MathNet.Numerics.Statistics

Is similar to Matlab and returns NaN. Does not throw.

Variance and Kurtosis both *Returns NaN if data has less than two (four) entries or if any entry is NaN.*[8].

[4]Matlab variance: https://se.mathworks.com/help/Matlab/ref/var.html

[5]Matlab kurtosis is not *excess* kurtosis and differs by 3 compared to scipy.stats.kurtosis

[6]Boost math_toolkit: url https://www.boost.org/doc/libs/1_76_0/libs/math/doc/html/-math_toolkit/univariate_statistics.html

[7]Gnu Scientific library: https://www.gnu.org/software/gsl/doc/html/statistics.html, and https://www.gnu.org/software/gsl/doc/html/err.html

[8]C# MathNet.Numerics.Statistics: Variance and Kurtosis: https://numerics.mathdotnet.com/api/MathNet.Numerics.Statistics/Statistics.htm#Variance, and https://numerics.mathdotnet.com/api/MathNet.Numerics.Statistics/Statistics.htm#Kurtosis

Returns NaN if data has less than four entries or if any entry is NaN.

4.5 R: MatrixStats

This library uses a special value to return an empty state, comparable to `std::optional` or `expected`.

If there are missing values in w that are part of the calculation (after subsetting and dropping missing values in x), then the final result is always NA of the same type as x.

4.6 Excel, Google Sheets

Transparently supports missing and degenerate data. For example, both `var` and kurtosis of a no data or a single number is `#DIV/0!`, which is then propagated further into calculations similar to how `NaN` and `Inf` works in C and C++.

4.7 Python: scipy.stats and numpy

In general, python uses exceptions a lot (for example on `"int("xyz")`), but still excess kurtosis (which nominally require four elements) results in

```
scipy.stats.kurtosis([]) is -3
scipy.stats.kurtosis([1]) is -3
scipy.stats.kurtosis([1,2]) is -2
scipy.stats.kurtosis([1,2,3]) is -1.5
scipy.stats.kurtosis([1,2,3,4]) is -1.36
```

Similarly,

```
numpy.mean([]) is NaN (with a runtime warning)
numpy.var([1]) is 0.0
numpy.var([]) is NaN (with a runtime warning)
```

4.8 Pandas, Stata and SAS

Python Pandas[9] deals with short inputs just as Matlab, Stata, Excel, Google sheets and reports NaN on for example variance of empty or a single element, and kurtosis of three elements. Kurtosis of four identical elements is reported as zero.

The exception is the python *statistics* module, which is similar to p1708r4 and throws a one-for-all exception on errors.

Stata: I could not find a reference for Stata, but it seems to be similar to Pandas but instead of (or in addition to?) NaN it has a special dot value.

SAS, as Pandas (and Stata?) explicitly documents that the standard deviation of less than two values is a special *null* value. Similarly, for kurtosis of all-identical values, the result is specified to be *null*.[10].

SAS, Stata, Pandas have special handling of these empty values when they propagate. For example, SAS ignores *null* values.

[9]Pandas comparison with Stata: https://pandas.pydata.org/docs/getting_started/comparison/comparison_with_stata.html

[10]SAS(R) 9.4 FedSQL Language Reference, Third Edition https://support.sas.com/documentation/cdl/en/fedsqlref/67364/HTML/default/viewer.htm#n00smq9b9h4mwhn1uwpp6v7gkt3m.htm, https://support.sas.com/documentation/cdl/en/fedsqlref/67364/HTML/default/viewer.htm#n0jnphh1mottc7n1f5x88lulngbb.htm

4.9 `std::log`, `std::sqrt` and `std::pow`

Just as `std::log`, `std::sqrt` and `std::pow`, the new functions should be specified so that they return as useful information as possible:

> (2) If no errors occur, the `variance` is returned.
>
> If a domain error occurs, an implementation-defined value is returned (NaN where supported)
>
> If a pole error or a range error due to overflow occurs, ±HUGE_VAL, ±HUGE_VALF, or ±HUGE_VALL is returned.[11]

If the implementation supports IEEE floating-point arithmetic (IEC 60559), Inf is returned instead of HUGE_VAL.

4.10 Conclusion

Degenerate cases due to lack of data can be considered explicit domain errors and return NaN (or implementation-defined value lacking ieee). A possibly alternative is to consider them pole errors, returning Inf.

This follows the conservative trail of SAS and lets kurtosis of all-identical values specified to NaN.

5 Other benefits of removing exceptions from the proposed Simple Statistical Functions

In addition to the arguments already presented, it should be noted that a large fraction of C++ developers can not use exceptions throughout their code: In the C++ Developer Survey "Lite"[12] about 45% of the respondents. It is a safe bet that the figure is significantly larger for users and shops that perform mathematical and statistical calculations.

By not throwing, it is also possible to explore making them *freestanding* in the sense of "C++ Freestanding and Conditionally Supported": Freestanding Roadmap P2268R0. Until then, and in practice, it opens up the possibility for partial C++ environment like CUDA to provide the functions.

6 Notes on Non-floating point input

All libraries mentioned in 1 support integer input. p1708r4 handles this by special casing the resulting type of non-integer projections. For example, the return type of `mean` is

```
std::conditional<
std::is_integral_v<
typename std::projected<std::ranges::iterator_t<R>, P>::value_type>,
double,
typename std::projected<
std::ranges::iterator_t<R>, P>::value_type>::type>
```

[11] `std::pow` also has: *If a range error occurs due to underflow, the correct result (after rounding) is returned.*

[12] C++ Developer Survey "Lite" 2018, 2020: https://isocpp.org/files/papers/CppDevSurvey-2020-04-summary.pdf, https://isocpp.org/files/papers/CppDevSurvey-2018-02-summary.pdf

This is similar to how `std::log` treats the integer case: The integer input: `double log ( IntegralType arg )`: *A set of overloads or a function template accepting an argument of any integral type. Equivalent to 2) (the argument is cast to double)..*

Similarly, the *mathematical special functions* such as `std::expint` treats integer arguments as if the input is converted to double, and returns a double.

Special math functions such as `std::beta` that take more than one argument also does *If any argument is long double, then the return type is also long double, otherwise the return type is always double.*

It would be beneficial to follow the example of existing mathematical functions and specify that **the result of integer input is the same as converting the input to double**, in addition to the currently specified return value.

6.1 Python and C++: generic algorithms and duck typing versus floating point arithmetic

It's interesting to consider python, since its duck-typing (dynamically but strongly typed) has many similarities with C++ templates. Python supports generics in the Stepanov sense but concept violations are discovered at run time.

As an example let's consider Python `numpy.var`. By default, the type used in the *internal computation* is *deduced* to be the input type, with special case handling of integers: "Type to use in computing the variance. For arrays of integer type the default is float64; for arrays of float types it is the same as the array type."

This matches the conclusions above.

6.2 Non-Integer, non-floating point input

Is not supported by p1708r4, but also not by existing functions such as `std::log`, `std::sqrt` and `std::pow`.

7 Conclusion

Math functions are susceptible to poles, overflow and degenerate cases which are sometimes not practically possible to detect until the implementation runs in to them. Implementations of the proposed mathematical functions will naturally encounter cases like overflow and floating point division by zero.

Users should expect that these cases can occur. On the other hand it's unreasonable for users to pre-analyse the data to predict degeneracies. Instead, users should be able to transparently calculate the statistical functions on unknown data sets and each call should produce a results which can be stored or processed further. This is in line with how simple (and "special") mathematical functions as well as user facing statistical tools operate.

Follow the lead of existing mathematical functions

By handling special conditions similarly to existing mathematical functions the need for the `stats_error` exception is eliminated and we return NaN in all[13] cases proposed to throw in p1708r4.

[13] Again, median and quantiles are excluded from the discussion.

Be conservative

In this approach we end up returning NaN on mean of empty data; returning NaN kurtosis of thee values just as Matlab, SAS, python:Pandas, Stata, Excel; R:MatrixStats, C# MathNet.Numerics.Statistics. Returning zero on poles is ambiguous and must be avoided.

Treat integers as double

Section 6.1 and 6 concludes that the result of integer computation should be specified to be that of converting the input input to double.

More details

It would also be beneficial to specify in as much details as possible the expectation and results with NaN and Inf input, as well as internal overflow[14] even for normal (as in `std::isnormal`) data. It's worth noting that a simple implementation of most proposed functions would include use of `std::pow` which guide the specification. It has (in ieee) intricate and thought through handling of special cases with best-effort results such as `pow(1, exp) == 1` even for NaN. `https://en.cppreference.com/w/cpp/numeric/math/pow` .

8 Appendix: Division by zero – NaN and exceptions in C#, Java, Python

For reference, these languages use exceptions [15,16] to report pole errors for integer division.

But in C#: *Floating-point arithmetic overflow or division by zero never throws an exception, because floating-point types are based on IEEE 754 and so have provisions for representing infinity and NaN (Not a Number).*.

Python is more keen on using exceptions and throws *ZeroDivisionError* even for the built in floats, but `numpy.float32` returns Inf (with a runtime warning that users can subscribe to, quite similar to C99 ieee).

[14]overflow: For example, Python numpy, the variance of [1.5e155,-1.5e155] overflows with numpy.var, returning Inf.

[15]C# and Java division by zero and exceptions: https://docs.microsoft.com/en-us/dotnet/api/system.dividebyzeroexception?view=net-5.0, https://docs.oracle.com/javase/8/docs/api/java/lang/ArithmeticException.html, https://www.baeldung.com/java-division-by-zero

[16]Java: 15.17.2. Division Operator: https://docs.oracle.com/javase/specs/jls/se16/html/jls-15.html#jls-15.17.2